NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ITERATIVE ALGORITHMS FOR TWO-PERSON ZERO-SUM GAMES

by

Piya Limsakul

March 1999

Thesis Advisor:	Alan R. Washburn
Thesis Co-advisor:	James N. Eagle

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 1999	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE ITERATIVE ALGORITHMS FOR TWO-PERSON ZERO SUM GAMES	**5. FUNDING NUMBERS**
6. AUTHOR(S) Piya Limsakul	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(maximum 200 words)*

In 1951, G.W. Brown proposed an iterative algorithm called 'fictitious play' for solving two-person zero-sum games. Although it is an effective method, the fictitious play algorithm converges slowly to the value of the game. Recently, Gass, Zafra, and Qiu proposed a modification that applies to symmetric games, i.e., games with skew-symmetric payoff matrices. To solve non-symmetric games via their modification, the games must be made symmetric via a transformation.

Gass, Zafra, and Qiu reported that their modified algorithm converges faster than the original fictitious play on a collection of randomly generated games. However, their results on non-symmetric games only apply to games whose values are near zero. When game values are far away from zero, this thesis empirically shows that the original fictitious play algorithm can outperform the modified one.

Gass, Zafra, and Qiu's method is static, in that the symmetric transformation is done once prior to the start of their modified algorithm. However, they suggested the exploration of dynamic methods where the transformation is periodically revised. This thesis proposes and investigates the convergence behavior of one dynamic transformation technique for solving general two-person zero-sum games.

14. SUBJECT TERMS Two-person Zero-sum Games, Regular Fictitious Play, Modified Fictitious Play, Symmetric Games, Game Value, Randomized Strategies	**15. NUMBER OF PAGES** 59
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFI- CATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

ITERATIVE ALGORITHMS FOR TWO-PERSON ZERO-SUM GAMES

Piya Limsakul
Lieutenant, Royal Thai Navy
B.S., Royal Thai Naval Academy, 1994

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the
NAVAL POSTGRADUATE SCHOOL
March 1999

Author: [illegible]
Piya Limsakul

Approved by: [illegible]
Alan R. Washburn, Thesis Advisor

[illegible]
James N. Eagle, Thesis Co-advisor

Siriphong Lawphongpanich
Siriphong Lawphongpanich, Second Reader

Richard E. Rosenthal
Richard E. Rosenthal, Chairman
Department of Operations Research

ABSTRACT

In 1951, G.W. Brown proposed an iterative algorithm called 'fictitious play' for solving two-person zero-sum games. Although it is an effective method, the fictitious play algorithm converges slowly to the value of the game. Recently, Gass, Zafra, and Qiu proposed a modification that applies to symmetric games, i.e., games with skew-symmetric payoff matrices. To solve non-symmetric games via their modification, the games must be made symmetric via a transformation.

Gass, Zafra, and Qiu reported that their modified algorithm converges faster than the original fictitious play on a collection of randomly generated games. However, their results on non-symmetric games only apply to games whose values are near zero. When game values are far away from zero, this thesis empirically shows that the original fictitious play algorithm can outperform the modified one.

Gass, Zafra, and Qiu's method is static, in that the symmetric transformation is done once prior to the start of their modified algorithm. However, they suggested the exploration of dynamic methods where the transformation is periodically revised. This thesis proposes and investigates the convergence behavior of one dynamic transformation technique for solving general two-person zero-sum games.

TABLE OF CONTENTS

I. INTRODUCTION

When it was proposed in 1951, Brown's fictitious play algorithm was not known to converge, but it was applied to a few two-person zero-sum (TPZS) games with success. Robinson [Ref. 1] later proved its convergence. Despite this result, the algorithm is not always the method of choice for solving TPZS games. In their book, Szép and Forgó [Ref. 2] state that 'Computational experience available up to now indicates that, for the solution of general matrix games, linear programming is the most efficient method.' However, the fictitious play algorithm is not impractical. For some large games or games where it is impossible or impractical to enumerate all possible strategies a priori (see, e.g., Ref. [3]), the fictitious play algorithm may be the only effective choice.

Recently, Gass, Zafra, and Qiu (GZQ) [Ref. 4] proposed a modification to the fictitious play algorithm. The modification assumes that the game is symmetric. Since non-symmetric games can be transformed into symmetric ones (see, e.g., Ref. [5]), their modification also applies to non-symmetric games. In their paper, GZQ also report that their modification converges faster than the original fictitious play algorithm. Their results are based on random games with 100×100 payoff matrices whose elements are Uniform random numbers between –100 and 100.

Table 1.1: Values of Games with Uniform[-100,100] Payoffs

Size	Density	Game values among a sample of 50 games		
		minimum	average	maximum
100 x 100	20%	-1.4970	0.1402	1.9367
100 x 100	40%	-1.1616	0.2806	2.2304
100 x 100	60%	-2.1494	0.0206	2.3894
100 x 100	80%	-1.8568	0.0667	2.3511
100 x 100	100%	-2.0555	0.1566	2.7096

Table 1.1 summarizes the values of some games with Uniform [-100,100] payoffs. In the table, the size of the payoff matrices is 100×100, i.e., each player has 100 strategies available, and the density is varied from 20% to 100%. For each combination of size and density, 50 random games are generated and solved by the simplex method (see, e.g., Ref. [6]) to find the game values. The minimum, average, and maximum game values for each sample of 50 games are reported in Table 1.1. They indicate that the values of the games with Uniform [-100,100] payoffs are near zero. Thus, GZQ tests are mainly on games with values that are near zero.

One objective of this thesis is to consider a larger class of random games and numerically compare the original against the modified fictitious play algorithm as proposed by GZQ. In addition, GZQ's modification is static, in that the symmetric transformation is performed once prior to start of their algorithm. This thesis also proposes an alternate modification in which the transformation is periodically updated with a different scaling parameter. The numerical results reported here indicate that this new modification converges more rapidly than both the original and modified fictitious play procedures proposed by GZQ.

To make the thesis self-contained, Chapter II describes symmetric and non-symmetric TPZS games. Chapter III discusses the existing techniques for solving these games and proposes an alternate modification for the fictitious play algorithm. In Chapter IV, variations of the fictitious play algorithm are compared against the original. Finally, Chapter V summarizes the results.

II. TWO-PERSON ZERO-SUM GAMES

A. DEFINITION

A two-person zero-sum (TPZS) game is a situation where there are two decision-makers (players) having directly opposite interests. In a TPZS game, one player, P1, has m strategies available and the other, P2, has n strategies. The outcome of the game depends on the strategy used by each player. If P1 and P2 choose the i^{th} and j^{th} strategies, respectively, then the outcome of the game, denoted as a_{ij}, represents the amount P2 has to pay P1. In other words, the payoffs to P1 and P2 are a_{ij} and $-a_{ij}$, respectively. Note that the sum of the two payoffs is zero, which explains the name of the game.

A TPZS game is completely defined when the payoff for each pair of P1 and P2 strategies is determined. These payoff can be summarized as an $m \times n$ matrix, generally referred to as a 'payoff matrix', i.e.,

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

In playing the game, both players are assumed to choose the best strategy to achieve the most favorable outcome. This means that P1, the player receiving the payoff a_{ij}, would choose the strategy that maximizes the outcome of the game. On the other hand, P2, having to pay a_{ij}, would choose the strategy that minimizes the outcome instead.

When a TPZS game has an equilibrium point, each player can guarantee an outcome of the game by always choosing one particular strategy. When equilibrium can not be achieved, both players are assumed to randomize their strategies. In other words, P1 and P2 choose strategies i and j, respectively, independently with probability x_i and y_j. The randomized strategies, $x = (x_1, \ldots, x_m)^T$ and $y = (y_1, \ldots, y_n)^T$, must satisfy the following conditions:

i) $0 \le x_i \le 1$ for all $i = 1, \ldots, m$

ii) $0 \le y_j \le 1$ for all $j = 1, \ldots, n$

iii) $\sum_{i=1}^{m} x_i = 1$

iv) $\sum_{j=1}^{n} y_j = 1$

To choose the best randomized strategy, P1 must solve the following optimization

$$\max_x \{ \min_j [\sum_{i=1}^{m} x_i a_{ij}]: \sum_{i=1}^{m} x_i = 1,\ x_i \ge 0, i = 1, \ldots, m \} \tag{2.1}$$

Similarly, P2 must solve the following problem to obtain the best randomized strategy

$$\min_y \{ \max_i [\sum_{j=1}^{n} a_{ij} y_j]: \sum_{j}^{n} y_j = 1,\ y_j \ge 0, j = 1, \ldots, n \} \tag{2.2}$$

Let x^* and y^* denote the optimal strategies for P1 and P2. Then, $v^* = (x^*)^T A y^*$ is the value of the game. In addition, it is well-known (see, e.g., Ref. [6]) that

$$v^* = \max_x \{ \min_j [\sum_{i=1}^{m} x_i a_{ij}]: \sum_{i=1}^{m} x_i = 1,\ x_i \ge 0, i = 1, \ldots, m \}, \text{ and}$$

$$v^* = \min_y \{ \max_i [\sum_{j=1}^{n} a_{ij} y_j]: \sum_{j}^{n} y_j = 1,\ y_j \ge 0, j = 1, \ldots, n \}$$

B. SYMMETRIC GAMES

A TPZS game is said to be *symmetric* if its payoff matrix is skew-symmetric, i.e., $A = -A^T$ or $a_{ij} = -a_{ji}$ for all i and j. The value of a symmetric game is always zero (see

[Ref.7]). In addition, the optimal randomized strategies for P1 and P2 are the same, i.e., $x^* = y^*$.

C. NON-SYMMETRIC GAMES

When the payoff matrix is not skew-symmetric, the game is *non-symmetric* and the value of the game may be nonzero. However, it is possible to transform a non-symmetric game into a symmetric one. In the literature, there are two transformation techniques, one proposed by von Neuman and the other by Gale, Kuhn and Tucker (GKT) [Ref. 5]. Among these two techniques, the latter is more attractive computationally, for the resulting symmetric game has a much smaller payoff matrix.

Given a payoff matrix A, the GKT technique defines the following payoff matrix

$$S = \begin{bmatrix} 0 & A & -c_1 \\ -A^T & 0 & c_2 \\ c_1^T & -c_2^T & 0 \end{bmatrix}$$

Where c_1 and c_2 are vectors in R^m and R^n, respectively. Furthermore, the components of both c_1 and c_2 are the same and they all are equal to δ, a positive constant. The 0's in S represent zero matrices of compatible dimension.

When the original payoff matrix A is of size $m \times n$, the resulting payoff matrix S is of size $(m+n+1) \times (m+n+1)$. Clearly, S is skew-symmetric. Let (p^*, q^*, λ^*) be an optimal randomized strategy for the symmetric game with payoff matrix S. If the original game has a positive value, it is possible to show (see [Ref. 5]) that $\sum_{i=1}^{m} p_i^* = \sum_{j=1}^{n} q_j^*$ and the optimal strategies for P1 and P2 in the original game are $x^* = \frac{1}{\mu} p^*$ and $y^* = \frac{1}{\mu} q^*$ where $\mu = \sum_{j=1}^{n} q_j^*$. In addition, the value of the original game is $\delta\lambda^* / \mu$.

For games with zero or negative values, it is possible to 'scale' or add a constant w to the payoff matrix A and obtain $A(w) = [a_{ij} + w]$. When w is sufficiently large, the resulting game has a positive value.

III. SOLVING TWO-PERSON ZERO-SUM GAMES

A. LINEAR PROGRAMMING

The problem for determining an optimal randomized strategy for P1 in equation 2.1 is equivalent to the following linear program (see, e.g., Ref. [6]):

OPT-1: maximize v

$$\text{subject to } \sum_i a_{ij} x_i \geq v \quad \text{for } j = 1, \ldots, n$$
$$\sum_i x_i = 1$$
$$x_i \geq 0 \quad \text{for } i = 1, \ldots, m$$

Similarly, the problem for determining an optimal randomized strategy for P2 in equation 2.2 is equivalent to the following linear program:

OPT-2: minimize w

$$\text{subject to } \sum_j a_{ij} y_j \leq w \quad \text{for } i = 1, \ldots, m$$
$$\sum_j y_j = 1$$
$$y_j \geq 0 \quad \text{for } j = 1, \ldots, n$$

It is easy to show that problems OPT-1 and OPT-2 are dual of each other. Moreover, if (v^*, x^*) and (w^*, y^*) are optimal to problems OPT-1 and OPT-2, respectively, then $v^* = w^*$.

B. REGULAR FICTITIOUS PLAY

To adopt the convention used in GZQ, the fictitious play algorithm as proposed by G. W. Brown in Ref. [8] is henceforth referred to as the regular fictitious play algorithm or RFP. To describe RFP, let A_i and A^j denote the i^{th} row and the j^{th} column of the payoff matrix A. As before, $x = (x_1, \ldots, x_i, \ldots, x_m)^T$ and $y = (y_1, \ldots, y_j, \ldots, y_n)^T$

represent the randomized strategies for P1 and P2, respectively. After k fictitious plays, define

i) $U(k) = [U_1(k), \ldots, U_i(k), \ldots, U_m(k)]^T$ as the cumulative payoff vector for P1,

ii) $a_i(k)$ as the number of times P1 uses strategy i in k plays,

iii) $V(k) = [V_1(k), \ldots, V_j(k), \ldots, V_n(k)]^T$ as the cumulative payoff vector for P2,

iv) $b_j(k)$ as the number of times P2 uses strategy j in k plays,

v) $m(k)$ as the lower bound for the value of the game, and

vi) $M(k)$ as the upper bound for the value of the game.

Below, RFP is stated with a stopping tolerance of $\varepsilon > 0$.

Regular Fictitious Play Algorithm (RFP)

Step 0: Set $U(0) = 0$, $V(0) = 0$, $a_i(0) = 0$ for all $i = 1, \ldots, m$, $b_j(0) = 0$ for all $j = 1, \ldots, n$, $m(0) = -\infty$, $M(0) = \infty$, and $k = 1$. Go to Step 1.

Step 1: Let $r = \arg\max_i \{U_1(k-1), \ldots, U_i(k-1), \ldots, U_m(k-1)\}$ (break ties arbitrarily). Set $V(k)^T = V(k-1)^T + A_r$ and $a_r(k) = a_r(k-1) + 1$. If $k > 1$, compute $M(k) = \min\{M(k-1), \frac{1}{(k-1)} U_r(k-1)\}$ and go to Step 2.

Step 2: Let $s = \arg\min_j \{V_1(k), \ldots, V_j(k), \ldots, V_n(k)\}$ (break ties arbitrarily). Set $U(k) = U(k-1) + A^s$ and $b_s(k) = b_s(k-1) + 1$. Compute $m(k) = \max\{m(k-1), \frac{1}{k} V_s(k)\}$ and go to Step 3.

Step 3: If $[M(k) - m(k)] \leq \varepsilon$, stop. The randomized strategy pair, $x = \frac{1}{k}(a_1(k), \ldots, a_m(k))^T$ and $y = \frac{1}{k}(b_1(k), \ldots, b_n(k))^T$, is approximately optimal. Otherwise, set $k = k + 1$ and go to Step 1.

In the first occurence Step 1, the r^{th} strategy (i.e, r^{th} row of the payoff matrix A) is arbitrarily assigned to P1 since $U(0)$ is a zero vector. In Step 2, P2 minimizes the cumulative payoff to P1 by choosing the s^{th} strategy. If the gap, i.e., the difference between the upper and lower bounds, in Step 3 is sufficiently small, the algorithm computes the resulting randomized strategies for both players and terminates.

To illustrate, consider the payoff matrix

$$A = \begin{bmatrix} 4 & 6 & 8 \\ 7 & 5 & 1 \end{bmatrix}.$$

Table 3.1 summarizes the first ten iterations of RFP. In Step 1, P1 arbitrarily chooses strategy 2 (or row 2) since $U(0) = 0$ and $V(1)$ is set to $(7, 5, 1)^T$. In Step 2, P2 then chooses strategy 3 (or column 3) since, against $V(1)$, P2 only has to pay 1 unit to P1 using this strategy. This sets $U(1) = (8, 1)^T$. The sequence of play just described is summarized in the first row ($k = 1$) of Table 3.1. The remaining rows are similarly obtained. In the fourth row ($k = 4$), $m(4)$ is the same as $m(3)$ since it is larger that $V_1(4)/4 = 4.75$. Similarly, in the ninth row ($k = 9$), $M(9) = M(8)$ since $M(8)$ is less than $U_2(k-1)/(k-1) = 44/8 = 5.5$

Table 3.1: Ten Iterations of the Regular Fictitious Play Algorithm

k	*s*	*U*(*k*-1)		*M*(*k*)	*r*	*V*(*k*)			*m*(*k*)
		i=1	*i*=2			*j*=1	*j*=2	*j*=3	
1	2	0	**0**	∞	2	7	5	**1**	1.0
2	3	**8**	1	8.0	1	11	11	**9**	4.5
3	3	**16**	2	8.0	1	**15**	17	17	5.0
4	1	**20**	9	6.7	1	**19**	23	25	5.0
5	1	**24**	16	6.7	1	**23**	29	33	5.0
6	1	**28**	23	5.6	1	**27**	35	41	5.0
7	1	**32**	30	5.3	1	**31**	37	49	5.0
8	1	36	**37**	5.3	2	**38**	42	50	5.0
9	1	40	**44**	5.3	2	**45**	47	51	5.0
10	1	44	**51**	5.3	2	52	**52**	52	5.2

C. MODIFIED FICTITIOUS PLAY

There are three variations of RFP discussed here. The first variation is called the modified fictitious play or MFP algorithm. This algorithm only applies to symmetric games. The other two are intended for non-symmetric games. Both use the GKT technique discussed in Chapter II to transform a non-symmetric game into a symmetric one. The variation described in GZQ, referred to here as 'Static MFP' or SMFP, performs the symmetric transforms once prior to the start of the algorithm. The remaining variation, 'Dynamic MFP' or DMFP, is new, Although, the basic idea is alluded to in GZQ. In this new variation, the transformation is periodically updated with new lower bounds on the value of the original game. As demonstrated in the next chapter, this variation converges faster on a collection of randomly generated games.

1. Modified Fictitious Play for Symmetric Games

The MFP algorithm for symmetric games is essentially the same as RFP. There is an additional step in MFP to take advantage of the fact that the value of a symmetric game is always zero and the optimal randomized strategies for P1 and P2 are the same. For an $n \times n$ payoff matrix, the MFP algorithm requires a switching interval K in addition to the stopping tolerance ε and it can be stated as follows.

Modified Fictitious Play Algorithm (MFP)

Step 0: Set $U(0) = 0$, $V(0) = 0$, $a_i(0) = 0$ for all $i = 1, \ldots, n$, $b_j(0) = 0$ for all $j = 1, \ldots, n$, $m(0) = -\infty$, $M(0) = \infty$, and $k = 1$. Go to Step 1.

Step 1: Let $r = \arg\max_i \{U_1(k-1), \ldots, U_i(k-1), \ldots, U_n(k-1)\}$ (break ties arbitrarily). Set $V(k)^T = V(k\text{-}1)^T + A_r$ and $a_r(k) = a_r(k-1) + 1$. If $k > 1$, then set $M(k) = \min\{M(k-1), \frac{1}{(k-1)}U_r(k-1)\}$. Go to Step 2.

Step 2: Let $s = \arg\min_j\{V_1(k), \ldots, V_j(k), \ldots, V_n(k)\}$ (break ties arbitrarily). Set $U(k) = U(k\text{-}1) + A^s$, $b_s(k) = b_s(k-1) + 1$, and $m(k) = \max\{m(k-1), \frac{1}{k}V_s(k)\}$. Go to Step 3.

Step 3: If $[M(k) - m(k)] \leq \varepsilon$, stop and either the randomized strategy $x = \frac{1}{k}(a_1(k),..., a_n(k))^T$ or $y = \frac{1}{k}(b_1(k),..., b_n(k))^T$ is approximately optimal. Otherwise, go to Step 4.

Step 4: If mod$(k, K) > 0$, set $k = k + 1$ and go to Step 1. Otherwise, set $\beta = \min\{|\min_j\{\frac{1}{k}V_j(k)\}|, |\max_i\{\frac{1}{k}U_i(k)\}|\}$. If $\beta = |\min_j\{\frac{1}{k}V_j(k)\}|$, then set $U(k) = - V(k)$ and $b(k) = a(k)$. Otherwise, set $V(k) = - U(k)$ and $a(k) = b(k)$. Set $k = k + 1$ and go to Step 1.

Steps 0 to 3 are essentially the same as those in RFP. The notation used in MFP reflects the fact that the payoff matrix, A, is square. In Step 4, MFP computes at every K iterations two game value estimates, $|\min_j\{\frac{1}{k}V_j(k)\}|$ and $|\max_i\{\frac{1}{k}U_i(k)\}|$. Depending on which estimate is closer to zero, the cumulative payoff vectors are 'switched' and both $a(k)$ and $b(k)$ are made equal to reflect the better of the two game value estimates. As empirically demonstrated in GZQ, the best value for K is one.

2. Modified Fictitious Play with Static Scaling

For a non-symmetric game, Static MFP or SMFP first transforms the $m \times n$ payoff matrix into the following skew-symmetric $(m + n + 1) \times (m + n + 1)$ payoff matrix:

$$S(w) = \begin{bmatrix} 0 & A(w) & -c_1 \\ A(w)^T & 0 & c_2 \\ c_1^T & -c_2^T & 0 \end{bmatrix}$$

where $A(w) = [a_{ij} + w]$ for some scaling factor $w > 0$ and the components of $c_1 \in R^m$ and and $c_2 \in R^n$ are all equal to $\delta > 0$. For the above transformation to be valid, the scaling factor w must be large enough to ensure that the payoff matrix, $A(w)$, yields a game with a positive value. In GZQ, the scaling parameter w is set to $|\min a_{ij}|+1$. Then, SMFP is essentially MFP applied to $S(w)$.

As before, let $S_r(w)$ and $S^s(w)$ denote the r^{th} row and the s^{th} column of the payoff matrix $S(w)$. Then, the Static MFP algorithm can be stated as follows.

Modified Fictitious Play Algorithm with Static Scaling (SMFP)

Step 0: Set $U(0) = 0$, $V(0) = 0$, $a_i(0) = 0$ for all $i = 1, \ldots, (m+n+1)$, $b_j(0) = 0$ for all $j = 1, \ldots, (m+n+1)$, $m(0) = -\infty$, $M(0) = \infty$, and $k = 1$. Go to Step 1.

Step 1: Let $r = \arg\max_i \{U_1(k-1), \ldots, U_i(k-1), \ldots, U_{(m+n+1)}(k-1)\}$ (break ties arbitrarily). Set $V(k)^T = V(k\text{-}1)^T + S_r(w)$ and $a_r(k) = a_r(k-1) + 1$. Go to Step 2.

Step 2: Let $s = \arg\min_j \{V_1(k), \ldots, V_j(k), \ldots, V_{(m+n+1)}(k)\}$ (break ties arbitrarily). Set $U(k) = U(k\text{-}1) + S^s(w)$ and $b_s(k) = b_s(k-1) + 1$. Go to Step 3.

Step 3: Set

$$x_i = a_i(k) \Big/ \sum_{i=1}^{m} a_i(k) \qquad \text{for } i = 1,\ldots, m,$$

$$y_j = a_{m+j}(k) \Big/ \sum_{j=1}^{n} a_{m+j}(k) \qquad \text{for } j = 1,\ldots,n,$$

$$p_i = b_i(k) \Big/ \sum_{i=1}^{m} b_i(k) \qquad \text{for } i = 1,\ldots, m, \text{ and}$$

$$q_j = b_{m+j}(k) \Big/ \sum_{j=1}^{n} b_{m+j}(k) \qquad \text{for } j = 1,\ldots,n.$$

Then, compute

$$m(k) = \max\{m(k-1), \min_i \sum_{j=1}^{n} A_{ij}(w)x_i, \min_i \sum_{j=1}^{n} A_{ij}(w)p_i\} \text{ and}$$

$$M(k) = \min\{M(k-1), \max_j \sum_{i=1}^{m} A_{ij}(w)y_j, \max_j \sum_{i=1}^{m} A_{ij}(w)q_j\}$$

If $[M(k) - m(k)] \leq \varepsilon$, stop and either the randomized strategy pair (x, y) or (p, q) is approximately optimal to the game with payoff matrix A. Otherwise, go to Step 4.

Step 4: If $\text{mod}(k, K) > 0$, set $k = k + 1$ and go to Step 1. Otherwise, set $\beta = \min\{|\min_j\{\frac{1}{k}V_j(k)\}|, |\max_i\{\frac{1}{k}U_i(k)\}|\}$. If $\beta = |\min_j\{\frac{1}{k}V_j(k)\}|$, then set $U(k) = -V(k)$ and $b(k) = a(k)$. Otherwise, set $V(k) = -U(k)$ and $a(k) = b(k)$. Set $k = k + 1$ and go to Step 1.

The main difference between MFP and SMFP is in Step 3 where the upper and lower bounds for the value of the game are calculated. The bounds in Step 3 of SMFP are for the original game, not the one with $S(w)$.

Step 3 may involve dividing by zero, but MATLAB, the numeric computational software in which the algorithm is implemented, handles this eventuality correctly (see appendix E).

3. Modified Fictitious Play with Dynamic Scaling

The results in Table 3.2 show that setting w to $|\min a_{ij}| + 1$ makes SMFP converge slowly. In fact, a better value for w is $-v$, where v is the value of the game. Table 3.2 displays the results on eight random games with 25%, 50%, 75%, and 100% density. Four games have 10×10 payoff matrices and the other have 20x20. Elements of all payoff matrices are uniformly distributed between -100 and 100. The stopping tolerance, ε, is 0.1 and the switching interval, K, is 1. The value of the game is obtained by solving the corresponding linear program discussed earlier.

Table 3.2: Numerical Results for MFP with two scaling parameters

Matrix Size	Matrix Density	Game value	$w = \lvert\min(a_{ij})\rvert + 1$		$w = -$ game value	
			CPU (sec)	Iteration	CPU (sec)	Iteration
10×10	25%	-18.58	24.27	6103	2.53	837
10×10	50%	-4.65	53.33	11778	41.75	9100
10×10	75%	-10.85	20.16	5278	13.45	3757
10×10	100%	-14.73	12.36	3496	6.81	2092
20×20	25%	1.22	49.04	9887	34.71	7657
20×20	50%	2.01	65.03	15813	30.93	6764
20×20	75%	1.61	74.26	16936	42.4	8426
20×20	100%	-2.80	56.80	11045	26.25	6059
		Total	355.25	80336	198.83	44692

In Table 3.2, the CPU times for MFP with $w = -v$ are between 10% to 80% of those for MFP with $w = \lvert\min a_{ij}\rvert+1$. Over all eight games, there is a 44% reduction in CPU times when $w = -v$. A similar conclusion also holds for the number of iterations.

The above results motivate the idea of adjusting the scaling parameter periodically. To be valid, the resulting payoff matrix $A(w)$ must yield a positive game value. One method is to simply set w to the negative of the best lower bound, i.e., $w = -m(k)$.

Let L be the rescaling interval. Then, the Dynamic MFP algorithm or DMFP can be stated as follows:

Modified Fictitious Play Algorithm with Dynamic Scaling (DMFP)

Steps 0 - 3: Same as SMFP

Step 4: If mod(k, K) > 0, go to Step 5. Otherwise, set $\beta = \min\{\lvert\min_j\{\frac{1}{k}V_j(k)\}\rvert, \lvert\max_i\{\frac{1}{k}U_i(k)\}\rvert\}$. If $\beta = \lvert\min_j\{\frac{1}{k}V_j(k)\}\rvert$, then set $U(k) = -V(k)$ and $b(k) = a(k)$. Otherwise, set $V(k) = -U(k)$ and $a(k) = b(k)$ and go to Step 5.

Step 5: Set $k = k + 1$. If $\mathrm{mod}(k, L) > 0$, go to Step 1. Otherwise, set $w = -m(\mathrm{k})$, recompute $S(w)$, and go to Step 1.

IV. NUMERICAL RESULTS

A. DATA GENERATION

To compare the algorithms discussed in Chapter III, random games with 100×100 payoff matrices are generated. For symmetric games, the payoffs, a_{ij}, are uniformly distributed between –100 and 100 (see appendix A). For non-symmetric games, three groups of payoff matrices are considered. In Group 1, the nonzero payoffs are uniformly distributed between –100 and 100. For Group 2, they are between –200 and 0. Finally, a_{ij}'s for Group 3 are between 0 and 200.

To generate payoff matrices with density θ, where $0 < \theta \leq 1$, a Uniform random number, μ_{ij}, between 0 and 1 is generated for each pair of strategies (i, j). If $\mu_{ij} \leq \theta$, then a Uniform random number is generated for a_{ij}. Otherwise, $a_{ij} = 0$. The games in Groups 1, 2, and 3 differ by a constant if and only if $\theta = 1$.

B. PARAMETER VALUES

All algorithms are terminated when the gap is less than or equal to 0.1, i.e., the stopping tolerance, ε, equals 0.1. As suggested by GZQ, the switching interval K is 1 for MFP, SMFP, and DMFP.

For SMFP, δ is set to $0.75(\max(a_{ij})-\min(a_{ij}))$. Recall that the parameter δ is the constant used in defining c_1 and c_2 for the symmetric transformation. In our preliminary study, other values for δ, e.g., $0.25(\max(a_{ij})-\min(a_{ij}))$, $0.50(\max(a_{ij})-\min(a_{ij}))$, $(\max(a_{ij})-\min(a_{ij}))$, and 1, did not show significant improvement in CPU time.

For games in Groups 1 and 2, the scaling parameter w is $|\min(a_{ij})| + 1$. Since games in Group 3 already have positive values, w is simply set to zero.

The parameter settings for DMFP are the same as those for SMFP. The additional parameter for the dynamic version is the rescaling interval, L, which is set at 5000.

C. IMPLEMENTATION

All four algorithms were implemented as MATLAB (Version 5.0) functions. The programs for all four functions and game generation are listed in the appendices. The reported CPU times in the following sections were generated using a 300 MHz Pentium II personal computer with 64 MEG of RAM.

D. SYMMETRIC GAMES

The results in Table 4.1 show that MFP outperforms RFP on four symmetric games from Group 1 with various densities.

Table 4.1: Computational Results for Symmetric Games in Group 1

Game	Den.	Method	Gap	Lower bound	Upper bound	CPU (sec)	Iterations $(\times 10^6)$
1	25%	RFP	0.0999	-0.0511	0.0488	476.75	0.1859
		MFP	0.1000	-0.0502	0.0498	40.37	0.0080
2	50%	RFP	0.0998	-0.0516	0.0482	849.97	0.2518
		MFP	0.1000	-0.0500	0.0499	63.55	0.0102
3	75%	RFP	0.0999	-0.0498	0.0502	919.34	0.3604
		MFP	0.0998	-0.0505	0.0493	83.37	0.0179
4	100%	RFP	0.1000	-0.0532	0.0468	795.32	0.3147
		MFP	0.0994	-0.0497	0.0497	79.92	0.0161

The CPU times for MFP in Table 4.1 are between 7.5% and 10.0% of those for RFP. Similarly, MFP also uses fewer iterations; they are between 4% and 5% of those for RFP.

Figure 4.1 displays a typical convergence behavior of RFP and MFP on symmetric games. MFP seems to converge directly to the solution. On the other hand, RFP has a long convergence tail. The success of MFP can be attributed to the switching of cumulative payoffs and strategies in Step 4, which takes full advantage of the special solution structure of symmetric games.

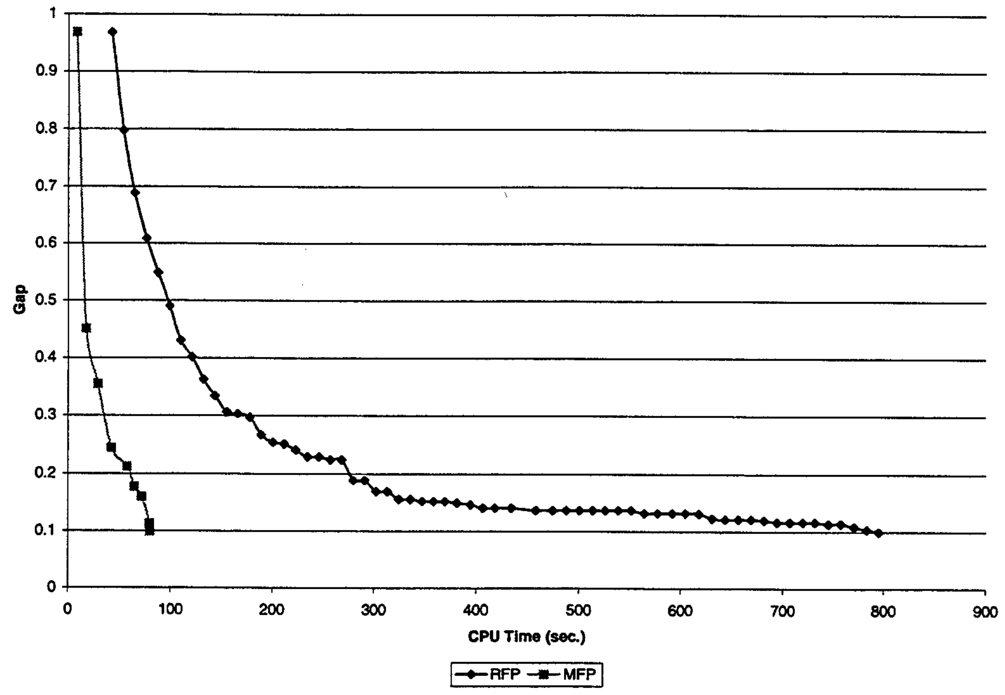

Figure 4.1: Convergence Behavior of RFP and MFP for a Symmetric Game in Group 1 with 100% Density

E. NON-SYMMETRIC GAMES

Two sets of experiments were performed, one set to compare RFP against SMFP and the other to compare RFP against DMFP. In these experiments, all three algorithms are terminated as soon as a 0.1 gap is achieved. As demonstrated below, DMFP is superior to RFP and RFP is superior to SMFP.

1. RFP and SMFP

Tables 4.2 to 4.4 summarize results for games in Groups 1, 2, and 3, respectively. Each table reports results on four random games, each with different densities. With the exception of game 3 in Table 4.4, RFP takes between 20% to 75% less time than SMFP to achieve a gap of 0.1 or better. For game 3 in Table 4.4, the CPU time of RFP is approximately 3% more than that of SMFP. In general, SMFP requires fewer iterations.

However, one iteration of SMFP is much more computationally intensive than one iteration of RFP.

Table 4.2: Computational Results for Games in Group 1

Game	Den.	Method	Gap	Lower bound	Upper bound	CPU (sec)	Iterations ($\times 10^6$)
1	25%	RFP	0.0999	-0.7148	-0.6149	348.51	0.1939
		SMFP	0.0997	-0.7103	-0.6106	1466.79	0.2402
2	50%	RFP	0.1000	-0.2571	-0.1571	718.53	0.3526
		SMFP	0.0999	-0.2627	-0.1628	1617.66	0.2788
3	75%	RFP	0.0998	0.5192	0.6190	989.81	0.4943
		SMFP	0.0999	0.5136	0.6135	2247.34	0.2898
4	100%	RFP	0.0999	0.1176	0.2175	801.25	0.4249
		SMFP	0.0998	0.1217	0.2216	1795.85	0.2486

Table 4.3: Computational Results for Games in Group 2

Game	Den.	Method	Gap	Lower bound	Upper bound	CPU (sec.)	Iterations ($\times 10^6$)
1	25%	RFP	0.1000	-24.8046	-24.7046	787.74	0.5171
		SMFP	0.0997	-24.8027	-24.7029	2849.15	0.5316
2	50%	RFP	0.1000	-49.6928	-49.5929	702.11	0.4591
		SMFP	0.0999	-49.6847	-49.5847	2183.56	0.4056
3	75%	RFP	0.0999	-73.8200	-73.7201	1211.00	0.5717
		SMFP	0.0999	-73.8156	-73.7157	1578.72	0.3191
4	100%	RFP	0.1000	-97.7296	-97.6296	798.40	0.5140
		SMFP	0.0998	-97.7225	-97.6226	1398.89	0.2880

Table 4.4: Computational Results for Games in Group 3

Game	Den.	Method	Gap	Lower bound	Upper bound	CPU (sec)	Iterations ($\times 10^6$)
1	25%	RFP	0.0999	25.8992	25.9992	1063.85	0.6843
		SMFP	0.0999	25.8965	25.9964	2850.68	0.4885
2	50%	RFP	0.0999	48.6506	48.7506	977.62	0.6341
		SMFP	0.0995	48.6532	48.7528	1269.44	0.2354
3	75%	RFP	0.1000	73.4574	73.5574	1168.87	0.7626
		SMFP	0.0998	73.4502	73.5500	1133.66	0.2399
4	100%	RFP	0.0999	99.0131	99.1130	825.47	0.5287
		SMFP	0.0998	99.0190	99.1188	1568.51	0.2869

It is also interesting to note in Tables 4.3 and 4.4 that SMFP takes the most CPU time in reaching a 0.1 gap for the game with 25% dense payoff matrices.

Figures 4.2 to 4.4 graphically display a typical convergence behavior for RFP and SMFP for games in Groups 1, 2 and 3 when ε (the stopping tolerance) = 0.1. These figures show that RFP converges to the game value faster than SMFP in terms of CPU time for all three groups of games.

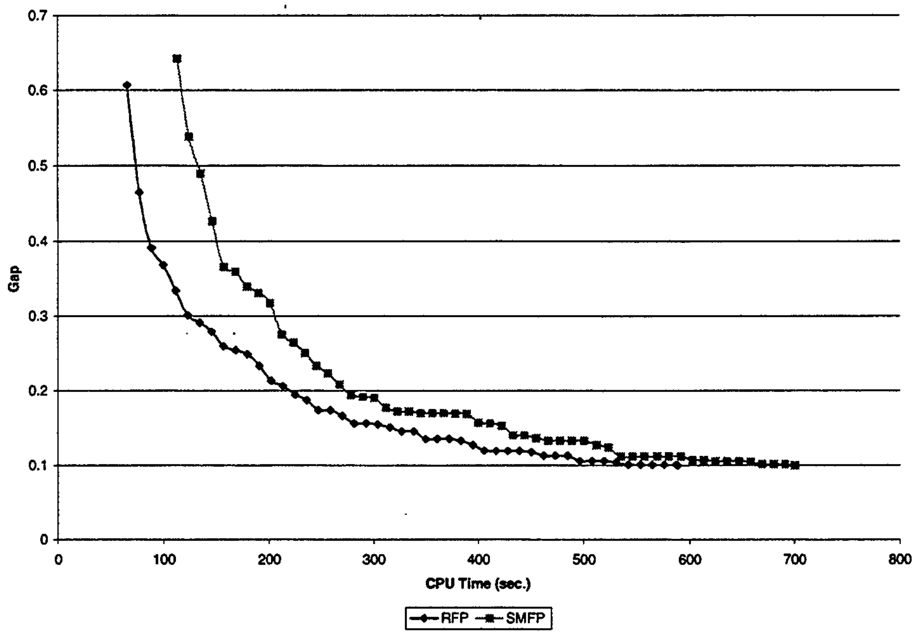

Figure 4.2: Convergence Behavior of RFP and SMFP for a Game in Group 1 with 50% Density

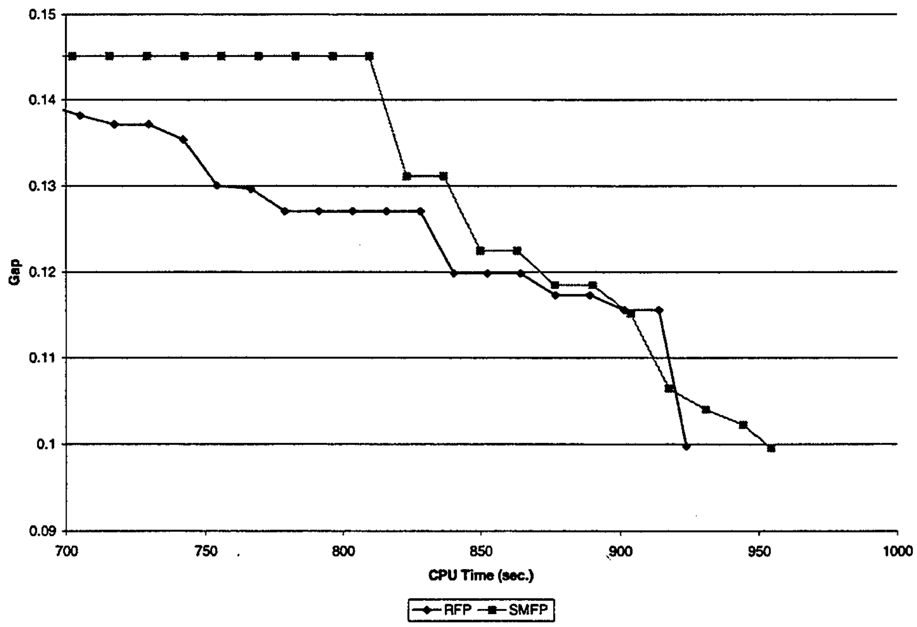

Figure 4.3: Convergence Behavior of RFP and SMFP for a Game in Group 2 with 50% Density

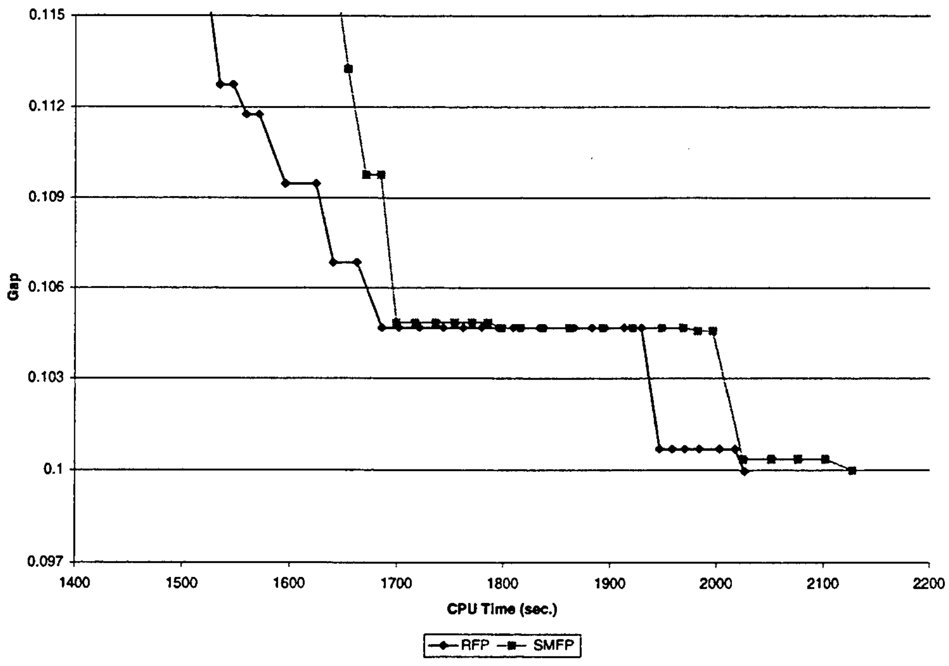

Figure 4.4: Convergence Behavior of RFP and SMFP for a Game in Group 3 with 50% Density

2. RFP and DMFP

As in the above subsection, Tables 4.5 to 4.7 summarize the computational results for games in Groups 1, 2, and 3, respectively. Each table reports results on four random games, each with different densities. In all 12 games, DMFP takes between 36% to 67% less time than RFP to achieve a gap of 0.1 or better. As with the static version, DMFP requires fewer iterations than RFP. However, unlike its counterpart, small numbers of DMFP iterations do not translate into more CPU seconds.

Table 4.5: Computational Results for Games in Group 1

Game	Den.	Method	Gap	Upper bound	Lower bound	CPU (sec)	Iterations ($\times 10^6$)
1	25%	RFP	0.0999	-0.7148	-0.6149	301.54	0.1939
		DMFP	0.0999	-0.7177	-0.6178	159.45	0.0228
2	50%	RFP	0.1000	-0.8660	-0.7660	461.92	0.2959
		DMFP	0.0994	-0.8634	-0.7640	255.68	0.0430
3	75%	RFP	0.1000	-0.3411	-0.2411	564.47	0.3576
		DMFP	0.1000	-0.3452	-0.2452	331.20	0.0495
4	100%	RFP	0.1000	0.1049	0.2048	617.20	0.3892
		DMFP	0.0992	0.1067	0.2059	395.30	0.0681

Table 4.6: Computational Results for Games in Group 2

Game	Den.	Method	Gap	Upper bound	Lower bound	CPU (sec)	Iterations ($\times 10^6$)
1	25%	RFP	0.10	-23.89	-23.79	906.16	0.5449
		DMFP	0.10	-23.89	-23.79	350.37	0.0547
2	50%	RFP	0.10	-49.42	-49.32	1029.64	0.6765
		DMFP	0.10	-49.42	-49.32	373.82	0.0663
3	75%	RFP	0.10	-71.59	-71.49	1036.88	0.5775
		DMFP	0.10	-71.58	-71.48	486.09	0.0771
4	100%	RFP	0.10	-100.02	-99.92	657.13	0.4192
		DMFP	0.10	-100.02	-99.93	378.05	0.0652

Table 4.7: Computational Results for Games in Group 3

Game	Den.	Method	Gap	Upper bound	Lower bound	CPU (sec)	Iterations ($\times 10^6$)
1	25%	RFP	0.10	23.05	23.15	800.48	0.5308
		DMFP	0.10	23.05	23.15	263.59	0.0401
2	50%	RFP	0.10	47.73	47.83	766.21	0.4984
		DMFP	0.10	47.74	47.84	423.58	0.0660
3	75%	RFP	0.10	75.27	75.37	978.94	0.6342
		DMFP	0.10	75.27	75.37	573.65	0.0898
4	100%	RFP	0.10	97.80	97.90	1111.86	0.7131
		DMFP	0.10	97.80	97.90	642.08	0.0774

Figures 4.5 to 4.7 graphically illustrate typical RFP and DMFP convergence behavior for games in Groups 1, 2 and 3, respectively. As before, ε, the stopping tolerance, is set at 0.1. As the numerical results suggest, DMFP outperforms RFP which, in turn, outperforms SMFP.

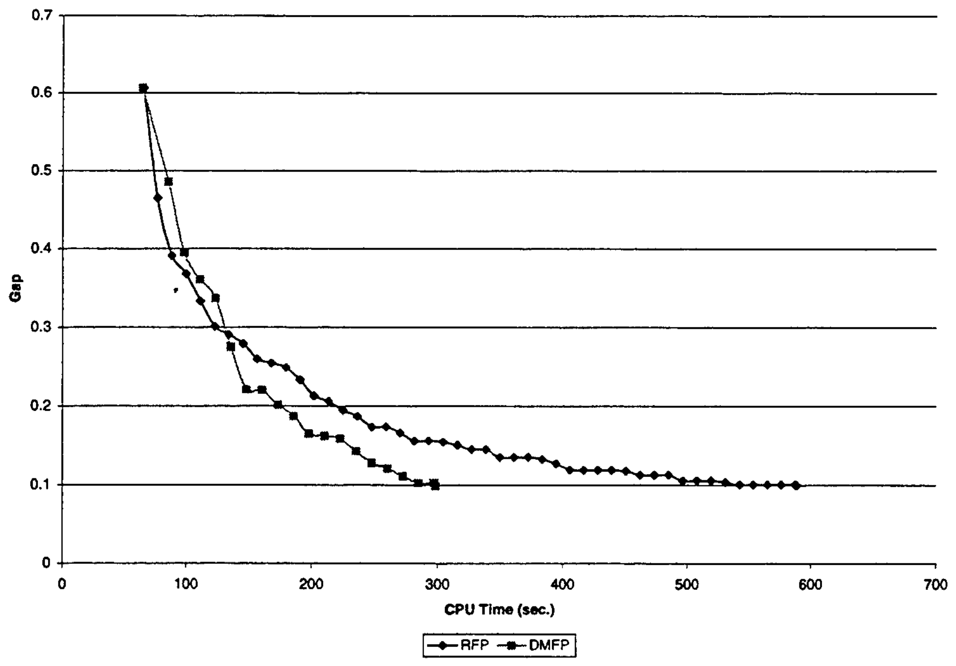

Figure 4.5: Convergence Behavior of RFP and DMFP for a Game in Group 1 with 50% Density

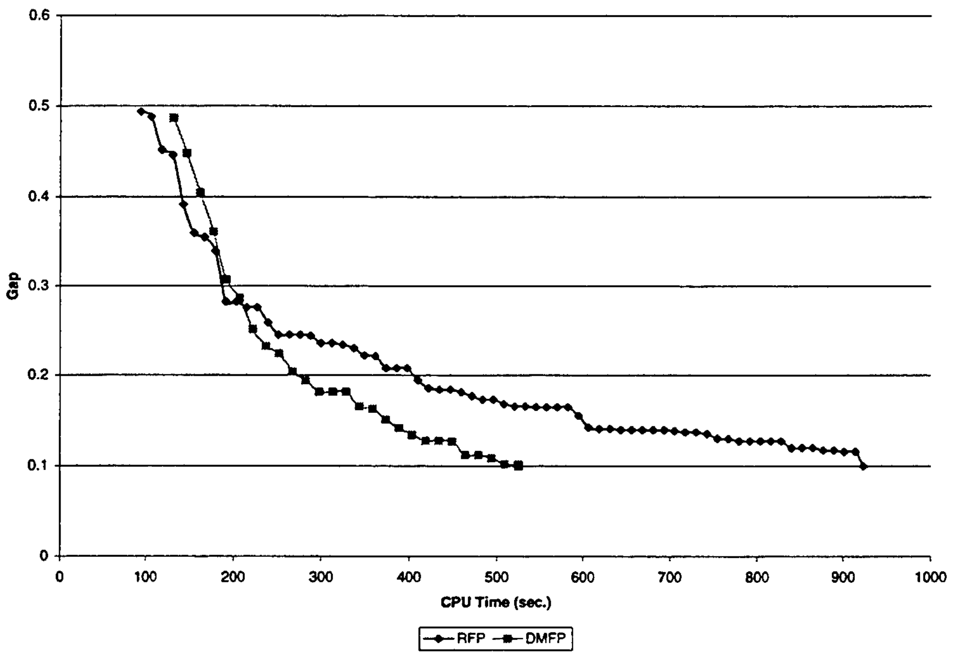

Figure 4.6: Convergence Behavior of RFP and DMFP for a Game in Group 2 with 50% Density

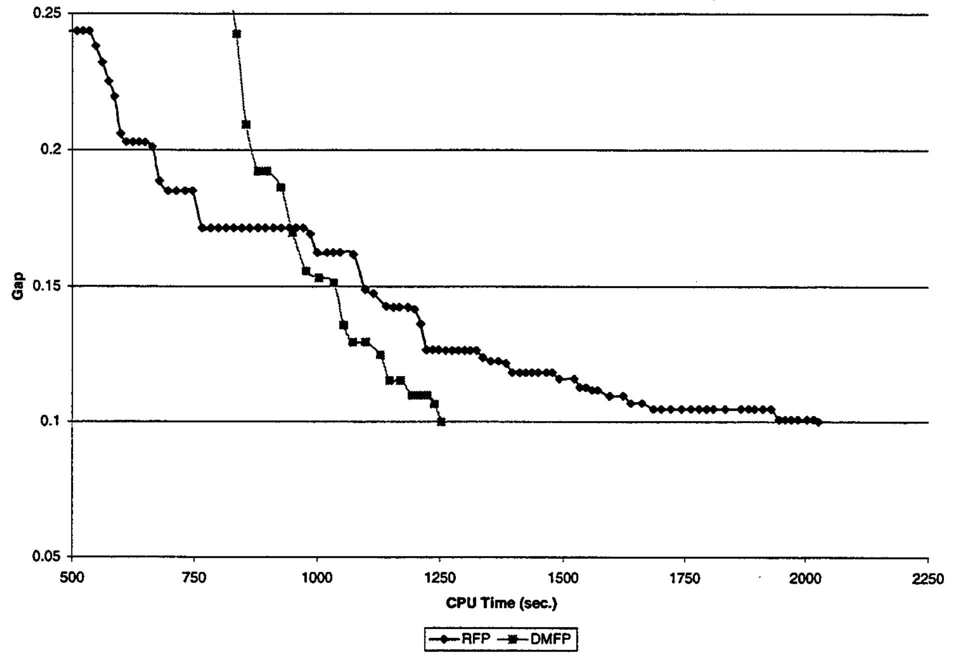

Figure 4.7: Convergence Behavior of RFP and DMFP for a Game in Group 3 with 50% Density

V. CONCLUSIONS AND SUGGESTION FOR FURTHER WORK

This thesis proposes an alternate modification to the fictitious play algorithm which can be considered as a dynamic variation of the one proposed by GZQ. The modification proposed here is dynamic, in that the symmetric transformation is updated periodically with a different scaling parameter. The results in Chapter IV indicate that this dynamic variation is computationally advantageous when compared to the original fictitious play or the static variation proposed by GZQ.

The results in Chapter IV also confirm that GZQ's modified fictitious play algorithm outperforms the original when applied to symmetric games. The results are reversed for non-symmetric games. The original fictitious play algorithm is rather robust and outperforms GZQ's static modification on random non-symmetric games.

Finally, this thesis also identifies several topics for further investigation. One topic is to investigate the choice of δ in defining c_1 and c_2 in the transformed matrix $S(w)$. The other is to investigate other choices for w, the scaling parameter, perhaps in conjunction with the choice of δ.

APPENDIX A. MATLAB CODES FOR GENERATING SKEW-SYMMETRIC MATRICES FOR SYMMETRIC GAMES

```
function A = skewSym(m,d)
% This function is used for generating an (m×m) skew-symmetric matrix
% with density d percent.
A = zeros(m,m);
for i = 1:m
  for j = 1:m
    if i < j
      if d > round(rand(1)*100) % Generate elements of A ~ U(-100,100)
        if 0.5 > rand(1)
          A(i,j) = ceil(rand(1)*100);
        else
          A(i,j) = floor(-100*rand(1));
        end
      end
    end
    if i > j
      A(i,j) = -A(j,i);
    end
  end
end
return
```

APPENDIX B. MATLAB CODES FOR GENERATING MATRICES FOR NON-SYMMETRIC GAMES

```
function A = density(m,n,d)
% This function generates an (m×n) general matrix with d % density
A = zeros(m,n); % A is a payoff matrix
for i = 1:m
  for j = 1:n
    if d > round(rand(1)*100) % Generate elements of A ~ U(-100,100)
      if 0.5 > rand(1)
        A(i,j) = ceil(rand(1)*100);
      else
        A(i,j) = floor(-100*rand(1));
      end
    end
  end
end
return
```

APPENDIX C. MATLAB CODES FOR RFP ALGORITHM

```
function [Upb,Lwb,gp,gap,k,compTime,reqflops,P1,P2] = RFP(gapneed,A)
% This function performs the RFP and gives the result whenever it achieves the
% needed gap

[m,n] = size(A);
V = zeros(1,n); % V is a P2's (choices)cumulative payoff vector
U = zeros(m,1); % U is a P1's (choices)cumulative payoff vector
x = zeros(m,1); % x is a P1's strategy vector
y = zeros(1,n); % y is a P2's strategy vector
r = 0;
mx = 10000;

t = cputime;
f = flops;

% The first iteration (k = 1)
first = ceil(rand(1)*m); % first is the i th row that P1 arbitrarily chooses
V = V + A(first,:);
x(first) = x(first) + 1;
lower(1) = min(V);

second = find(V==min(V)); % second is the j th column that P2 chooses to respond
if length(second) > 1
  nn = ceil(rand(1)*(length(second));
  second = second(nn);
end
U = U + A(:,second);
y(second) = y(second) + 1;
upper(1) = max(U);
gap(1) = upper(1) - lower(1);
k = 1;
ko = 1;

while  gap(ko) > gapneed
  k = k + 1;
  ko = ko + 1;
  st = find(U==max(U)); % st is the i th row giving max payoff that P1 expect
  if length(st) > 1
    nn = ceil(rand(1)*length(st));
    st = st(nn);
  end
  V = V +  A(st,:);
  x(st) = x(st) + 1;
  second = find(V==min(V)); % second is the j th column that P2 chooses to respond
  if length(second) > 1
    nn = ceil(rand(1)*length(second));
    second = second(nn);
  end
  U = U + A(:,second);
  y(second) = y(second) + 1;
  upper(ko) = max(U)/k;
  lower(ko) = min(V)/k;
  if upper(ko) > upper(ko-1)
```

```
    upper(ko) = upper(ko-1);
  end
  if lower(ko) < lower(ko-1)
    lower(ko) = lower(ko-1);
  end
  gap(ko) = upper(ko) - lower(ko);

  % In order to save the memory space,
  % the following loop controls the result vectors (lower, upper and gap)
  % to have at most mx = 10,000 elements inside.
  if mod(k-1,mx) == 0
    r = r + 1;
    disp([r, gap(ko-1)]) %display the gap at every 10,000 iterations
    lower(1) = lower(ko);
    lower = lower(1:ko-1);
    upper(1) = upper(ko);
    upper = upper(1:ko-1);
    gap(1) = gap(ko);
    gap = gap(1:ko-1);
    ko = 1;
  end
end
k;
if k > mx
  if ko < 5000
    lower = [lower((mx-5000+ko+1):mx),lower(1:ko)];
    upper = [upper((mx-5000+ko+1):mx),upper(1:ko)];
    gap = [gap((mx-5000+ko+1):mx),gap(1:ko)];
    ko = 5000;
  else
    lower = lower(1:ko);
    upper = upper(1:ko);
    gap = gap(1:ko);
  end
else
  lower = lower(1:ko);
  upper = upper(1:ko);
  gap = gap(1:ko);
end
P1 = x./k;
P2 = y./k;
Upb = upper(ko); Lwb = lower(ko);
gp = gap(ko);
reqflops = flops - f;
compTime = cputime - t;

return
```

APPENDIX D. MATLAB CODES FOR MFP ALGORITHM FOR SYMMETRIC GAMES

```
function [k,Upb,Lwb,gp,gap,compTime,reqflops] = MFPSym(gapneed,B)
% This function performs the MFP with an attempt to duplicate MFP algorithm in Gass,
% Zafra's paper.
% This function gives the result whenever it achieves the needed gap

K = 1;
r = 0; mx = 10000;
A = B; % input matrix game
[m,n] = size(A);
V = zeros(1,n); % V is a P2's (choices)cumulative payoff vector
U = zeros(m,1); % U is a P1's (choices)cumulative payoff vector
x = zeros(m,1); % x is a P1's strategy vector
y = zeros(1,n); % y is a P2's strategy vector
t = cputime;
f = flops;

% The first iteration (k = 1)
first = round(1+rand(1)*(m-1)); % first is the i th row that P1 arbitrarily chooses
V = V + A(first,:);
x(first) = x(first) + 1;
lower(1) = min(V);

second = find(V==min(V)); % second is the j th column that P2 chooses to respond
if length(second) > 1
  nn = round(1+rand(1)*(length(second)-1));
  second = second(nn);
end
U = U + A(:,second);
y(second) = y(second) + 1;
upper(1) = max(U);
gap(1) = upper(1) - lower(1);
k = 1;
ko = 1;

if K == 1
  if abs(max(U)) < abs(min(V))
    V = -(U');
  else
    U = -(V');
  end
end

while gap(ko) > gapneed
  k = k + 1;
  ko = ko + 1;
  st = find(U==max(U)); % st is the i th row giving max payoff that P1 expect
  if length(st) > 1
    nn = round(1+rand(1)*(length(st)-1));
    st = st(nn);
  end
  V = V +  A(st,:);
  x(st) = x(st) + 1;
```

```
    second = find(V==min(V)); % second is the j th column that P2 chooses to respond
    if length(second) > 1
      nn = round(1+rand(1)*(length(second)-1));
      second = second(nn);
    end
    U = U + A(:,second);
    y(second) = y(second) + 1;
    upper(ko) = max(U)/k;
    lower(ko) = min(V)/k;
    if upper(ko) > upper(ko-1)
      upper(ko) = upper(ko-1);
    end
    if lower(ko) < lower(ko-1)
      lower(ko) = lower(ko-1);
    end
    gap(ko) = upper(ko) - lower(ko);

    if mod(k,K) == 0
      if abs(max(U)/k) < abs(min(V)/k)
        V = -(U');
      else
        U = -(V');
      end
    end

  % In order to save the memory space,
  % the following loop controls the result vectors (lower, upper and gap)
  % to have at most mx = 10,000 elements inside.
  if mod(k-1,mx) == 0
      r = r + 1;
      disp([r, gap(ko-1)])  %display the gap at every 10,000 iterations
      lower(1) = lower(ko);
      lower = lower(1:mx);
      upper(1) = upper(ko);
      upper = upper(1:mx);
      gap(1) = gap(ko);
      gap = gap(1:mx);
      ko = 1;
    end
  end

  reqflops = flops - f;
  compTime = cputime - t;

  if k > mx
    if ko < 5000   %number of iterations is in [r*10000 , r*10000+5000)
      lower = [lower((mx-5000+ko+1):mx),lower(1:ko)];
      upper = [upper((mx-5000+ko+1):mx),upper(1:ko)];
      gap = [gap((mx-5000+ko+1):mx),gap(1:ko)];
      ko = 5000;
    else  %number of iterations is in [r*10000+5000 , (r+1)*10000)
      lower = lower(1:ko);
      upper = upper(1:ko);
      gap = gap(1:ko);
    end
  else %number of iterations < 10000
  lower = lower(1:ko);
  upper = upper(1:ko);
  gap = gap(1:ko);
end
Upb = upper(end); Lwb = lower(end); gp = gap(end);
return
```

APPENDIX E. MATLAB CODES FOR SMFP

```
function [count,Upb,Lwb,gp,gap,compTime,reqflops,P1,P2] = SMFP(gapneed,A,w)
% This function performs the MFP with an attempt to duplicate MFP algorithm in Gass,
% Zafra's paper.
% This function gives the result whenever it achieves the needed gap

r = 0;
mx = 10000;
k = 1;
[m,n] = size(A);
alpha = 0.75;
c = alpha*(max(max(A)) - min(min(A)));
An = A + w;
S = [zeros(m) An  -c*ones(m,1);
    -An' zeros(n) c*ones(n,1);
     c*ones(1,m) -c*ones(1,n) 0];
[sm,sn] = size(S);

U = zeros(sm,1);
V = zeros(1,sn);
x = zeros(sm,1);
y = zeros(1,sn);

t = cputime;
f = flops;

%start the first iteration
st = ceil(rand(1)*sm); % Row player randomly choose his row strategy
x(st) = x(st) + 1;
V = V + S(st,:);

nd = find(V == min(V));
if (length(nd) > 1)   % Randomly choose to break the tie
  i = ceil(rand(1)*length(nd));
  nd = nd(i);
end
y(nd) = y(nd) + 1;
U = U + S(:,nd);

%%%%%%%%%%%%%%%%%%%%%%%
xbar1 = x(1:m)./sum(x(1:m));                % This part will cause zero division errors
ybar1 = x(m+1:m+n)./sum(x(m+1:m+n));        % for some initial iterations, however MATLAB
xbar2 = y(1:m)./sum(y(1:m));                % can continue to the end of mission
ybar2 = y(m+1:m+n)./sum(y(m+1:m+n));

lower(1) = max(min(xbar1'*A), min(xbar2*A));
upper(1) = min(max(A*ybar1), max(A*ybar2'));
gap(1) = upper(1) - lower(1);

if k == 1
  if (abs(max(U)) < abs(min(V)))
    V = -U';
    x = y';
  else
```

```
      U = -V';
      y = x';
    end
  end
  count = 1;
  co = 1;

  while (gap(co) ~= 0)
    co = co + 1;
    count = count + 1;

    st = find(U == max(U));
    if (length(st) > 1)  % Randomly choose to break the tie
      i = ceil(rand(1)*length(st));
      st = st(i);
    end

    x(st) = x(st) + 1;
    V = V + S(st,:);

    nd = find(V == min(V));
    if (length(nd) > 1)  % Randomly choose to break the tie
      i = ceil(rand(1)*length(nd));
      nd = nd(i);
    end

    y(nd) = y(nd) + 1;

    U = U + S(:,nd);
    %%%%%%%%%%%%%%%%%

    xbar1 = x(1:m)./sum(x(1:m));
    ybar1 = x(m+1:m+n)./sum(x(m+1:m+n));
    xbar2 = y(1:m)./sum(y(1:m));
    ybar2 = y(m+1:m+n)./sum(y(m+1:m+n));

    lower(co) = max(min(xbar1'*A), min(xbar2*A));
    upper(co) = min(max(A*ybar1), max(A*ybar2'));
    if (lower(co) < lower(co-1))
      lower(co) = lower(co-1);
    end
    if (upper(co) > upper(co-1))
      upper(co) = upper(co-1);
    end

    gap(co) = upper(co) - lower(co);

    if (mod(count,k) == 0)
      if (abs(max(U)) < abs(min(V)))
        V = -U';
        x = y';
      else
        U = -V';
        y = x';
      end
    end
  if (gap(co) <= gapneed)
    break
  end

  % In order to save the memory space,
  % the following loop controls the result vectors (lower, upper and gap)
  % to have at most mx = 10,000 elements inside.
  if mod(count-1,mx) == 0
    r = r + 1;
   disp([r, gap(co-1)])  %display the gap at every 10,000 iterations
    lower(1) = lower(co);
    lower = lower(1:mx);
    upper(1) = upper(co);
    upper = upper(1:mx);
    gap(1) = gap(co);
    gap = gap(1:mx);
    co = 1;
  end
end

compTime = cputime - t;
reqflops = flops - f;


if count > mx
  if co < 5000   %number of iterations is in [r*10000 , r*10000+5000)
    lower = [lower((mx-5000+co+1):mx),lower(1:co)];
    upper = [upper((mx-5000+co+1):mx),upper(1:co)];
    gap = [gap((mx-5000+co+1):mx),gap(1:co)];
    co = 5000;
  else  %number of iterations is in [r*10000+5000 , (r+1)*10000)
    lower = lower(1:co);
    upper = upper(1:co);
    gap = gap(1:co);
  end
else %number of iterations < 10000
  lower = lower(1:co);
  upper = upper(1:co);
  gap = gap(1:co);
end

Upb = upper(end);
Lwb = lower(end);
gp = gap(end);
P1 = xbar1;
P2 = ybar2;

return
```

APPENDIX F. MATLAB CODES FOR DMFP

```
function [count,Upb,Lwb,gp,gap,compTime,reqflops,P1,P2] = DMFP(gapneed,A,w)
% This function performs the new modified version of MFP algorithm in Gass,
% Zafra's paper.
r = 0;
mx = 10000;
k = 1;
[m,n] = size(A);
alpha = 0.75;
c = alpha*(max(max(A)) - min(min(A)));
An = A + w;
S = [zeros(m) An  -c*ones(m,1);
  -An' zeros(n) c*ones(n,1);
  c*ones(1,m) -c*ones(1,n) 0];
[sm,sn] = size(S);

U = zeros(sm,1);
V = zeros(1,sn);
x = zeros(sm,1);
y = zeros(1,sn);

t = cputime;
f = flops;

%start the first iteration
st = ceil(rand(1)*sm); % Row player randomly choose his row strategy
x(st) = x(st) + 1;
V = V + S(st,:);

nd = find(V == min(V));
if (length(nd) > 1)   % Randomly choose to break the tie
  i = ceil(rand(1)*length(nd));
  nd = nd(i);
end

y(nd) = y(nd) + 1;

U = U + S(:,nd);
%%%%%%%%%%%%%%%%%%%%%%%%%%%

xbar1 = x(1:m)./sum(x(1:m));                 % This part will cause zero division errors
ybar1 = x(m+1:m+n)./sum(x(m+1:m+n));         % for some initial iterations, however MATLAB
xbar2 = y(1:m)./sum(y(1:m));                 % can continue to the end of mission.
ybar2 = y(m+1:m+n)./sum(y(m+1:m+n));

lower(1) = max(min(xbar1'*A), min(xbar2*A));
upper(1) = min(max(A*ybar1), max(A*ybar2'));
gap(1) = upper(1) - lower(1);

if k == 1
  if (abs(max(U)) < abs(min(V)))
    V = -U';
    x = y';
  else
```

```
    U = -V';
    y = x';
  end
end
count = 1;
co = 1;

while (gap(co) ~= 0)
  co = co + 1;
  count = count + 1;

  st = find(U == max(U));
  if (length(st) > 1)  % Randomly choose to break the tie
    i = ceil(rand(1)*length(st));
    st = st(i);
  end

  x(st) = x(st) + 1;
  V = V + S(st,:);

  nd = find(V == min(V));
  if (length(nd) > 1)  % Randomly choose to break the tie
    i = ceil(rand(1)*length(nd));
    nd = nd(i);
  end

  y(nd) = y(nd) + 1;

  U = U + S(:,nd);
  %%%%%%%%%%%%%%%%%%
  xbar1 = x(1:m)./sum(x(1:m));                 % This part will cause zero division errors
  ybar1 = x(m+1:m+n)./sum(x(m+1:m+n));         % for some initial iterations, however MATLAB
  xbar2 = y(1:m)./sum(y(1:m));                 % can continue to the end of mission.
  ybar2 = y(m+1:m+n)./sum(y(m+1:m+n));
  lower(co) = max(min(xbar1'*A), min(xbar2*A));
  upper(co) = min(max(A*ybar1), max(A*ybar2'));
  if (lower(co) < lower(co-1))
    lower(co) = lower(co-1);
  end
  if (upper(co) > upper(co-1))
    upper(co) = upper(co-1);
  end
  gap(co) = upper(co) - lower(co);
  if (mod(count,k) == 0)
    if (abs(max(U)) < abs(min(V)))
      V = -U';
      x = y';
    else
      U = -V';
      y = x';
    end
  end
  if (gap(co) <= gapneed)
    break
  end
  % In order to save the memory space,
  % the following loop controls the result vectors (lower, upper and gap)
  % to have at most mx = 10,000 elements inside.
  if mod(count-1,mx) == 0
    r = r + 1;
    %format long e ;
    disp([r, gap(co-1)])  %display the gap at every 10,000 iterations
    lower(1) = lower(co);
    lower = lower(1:mx);
    upper(1) = upper(co);
    upper = upper(1:mx);
    gap(1) = gap(co);
    gap = gap(1:mx);
    co = 1;
  end
% Dynamic scaling part
%%%%%%%%%%%%%%%%%%%%%%%%%%
  if (mod(count,5000) == 0) % dynamically update w every 5000 iterations
    w = -0.5*(upper(co) + lower(co));
    An = A + w;
     S = [zeros(m) An  -c*ones(m,1);
       -An'  zeros(n)  c*ones(n,1);
        c*ones(1,m) -c*ones(1,n) 0];
  end
%%%%%%%%%%%%%%%%%%%%%%%%%%%
end

compTime = cputime - t;
reqflops = flops - f;

if count > mx
  if co < 5000   %number of iterations is in [r*10000 , r*10000+5000)
    lower = [lower((mx-5000+co+1):mx),lower(1:co)];
    upper = [upper((mx-5000+co+1):mx),upper(1:co)];
    gap = [gap((mx-5000+co+1):mx),gap(1:co)];
    co = 5000;
  else  %number of iterations is in [r*10000+5000 , (r+1)*10000)
    lower = lower(1:co);
    upper = upper(1:co);
    gap = gap(1:co);
  end
else %number of iterations < 10000
  lower = lower(1:co);
  upper = upper(1:co);
  gap = gap(1:co);
end

Upb = upper(end);
Lwb = lower(end);
gp = gap(end);
P1 = xbar1;
P2 = ybar2;

Return
```

LIST OF REFERENCES

1. Robinson, J., "An iterative method of solving a game", *Annals of Mathematics*, 54, 296-301, 1951.

2. Szép, J. and F. Forgó. *Introduction to the Theory of Games*, D. Reidel Pub. Co. (Holland), 152-163, 1985.

3. Eagle, J.N. and Washburn, A.R., "Cumulative Search-Evasion Games", *Naval Research Logistics*, Vol. 38, pp. 495-510, 1991.

4. Gass, S.I., Zafra, P.M.R., and Qiu, Z., "Modified Fictitious Play", *Naval Research Logistics*, Vol. 43, pp. 955-970, 1996.

5. Gale, D., Kuhn, H.W., and Tucker, A.W., "On Symmetric Games." in H.W. Kuhn and A.W. Tucker (Eds.), "Contribution to the Theory of Games", *Annals of Mathematics Study*, No. 24, Princeton University Press, Princeton, NJ, Vol. 1, pp. 81-87, 1950.

6. Winston, W. I., *Operations Research Applications and Algorithms*, 2nd ed., PWS-KENT, Co., 1991.

7. Owen, G., "Two-Person Zero-Sum Games", *Game Theory*, 3rd ed., ACADEMIC PRESS, Inc., 1995.

8. Brown, G.W., "Iterative Solutions of Games by Fictitious Play." In T.C. Koopmans (ed.), *Activity Analysis of Production and Allocation*, Cowles Commission Monograph 13, 377-380, 1951.

INITIAL DISTRIBUTION LIST

No. of Copies

1. Defense Technical Information Center..2
   8725 John J. Kingman Rd. STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library..2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943

3. Library...1
   Institute of Advanced Naval Studies
   105 Mu 3 Salaya Rd, Salaya
   Puttamonton, Nakorn-Pathom 73170
   Thailand

4. Professor Alan R. Washburn, Code OR/Ws...............................2
   Department of Operations Research
   Naval Postgraduate School
   Monterey, CA 93943

5. Professor James N. Eagle, Code OR/Er...................................2
   Department of Operations Research
   Naval Postgraduate School
   Monterey, CA 93943

6. Professor Siriphong Lawphongpanich, Code OR/Lp......................1
   Department of Operations Research
   Naval Postgraduate School
   Monterey, CA 93943

7. Professor Saul I. Gass..1
   College of Business and Management
   University of Maryland
   College Park, MD 20742

8. Professor Pablo M.R. Zafra...1
   Mathematics Department
   Kean College
   Union, NJ 07083

9. Professor Ziming Qiu..1
   Mathematics Department
   University of Maryland
   College Park, MD 20742

10. LT Piya Limsakul..2
    1855/158 Jaransanithwong 75
    Bangkok 10700
    Thailand